ECE 331 Project 2

Machine Learning (Linear Regression), Matrix Multiplication, and Cache Simulation in MIPS assembly language

(Due Friday, November 22nd 11:59 PM)

Objectives:

- Learn weights for a linear regression function that predicts the performance of a processor based on empirical data
- Use learned weights in an MIPS assembly program to perform matrix multiplication on a matrix of test data
- Perform cache simulations to understand cache performance across different cache architectures

A Crash Course on Machine Learning and Linear Regression

In this project, you will be implementing a rudimentary supervised machine learning algorithm, called linear regression. In previous algebra courses, you have already encountered a simple form of linear regression where you are can fit a linear function to a set of values using the formula:

$$y(x) = mx + b \text{ (1)}$$

Where *m* is the slope of a line and *b* is the y-intercept. A more general form of Linear Regression is of the form:

$$y(x) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + d \text{ (2)}$$

Where there are *n* linear weights that get multiplied by *n* input terms and are summed together along with a bias value *d*.

When linear regression is used in the context of machine learning, we can think of the terms $x_1 \dots x_n$ as a set of input parameters for a particular example and weights

$w_1 \dots w_n$ and bias term *d* as a learned set of weights that are used to predict the output *y*.
After determining the weights for the linear function defined above, to determine the predicted output *y* for a sequence of input examples *x*, we can formulate this computation as a matrix multiplication:

$$Y = W \cdot X + d \ (3)$$

Where *W* is a 1 x *n* matrix and *X* is an *n* x *m* matrix, *Y* is a 1 x *m* matrix of output predictions, and *d* is a simple scalar value. The dot denotes the dot product operation, which is the operation performed in equation 2 above.

There a number of algorithms that can be used to estimate the best values for these weights such as ordinary least squares or gradient descent, but a study of these techniques is outside of the scope of this course. The goal of these techniques is to minimize the error of predictions after fixing the weight parameters. In this project, we will be using the tool Weka to learn these weights for a dataset that maps CPU parameters to ranked processor performance.

To learn the weights needed for our matrix multiplication, the first step is to download the open source data mining tool Weka:

https://www.cs.waikato.ac.nz/ml/weka/downloading.html

Weka is a data mining tool that can be used to learn machine learning model representations of data. It is a powerful tool, but we will be using it to perform a simple linear regression. In order to use Weka for machine learning tasks, it must be provided a set of training examples that use the .arff file extension. This is similar to a comma separated value format file (.csv), but has extra header information that describes the format of the input data and whether to consider each parameter a discrete set of classes, or continuous values that are integer or floating point values.

For linear regression, we need a .arff file that uses a set of continuous values that give us input values *x* and the actual output value *y*. The data for this project is found on Piazza under 'General Resources' and is the file `cpu.arff`.

If opened in a text editor, we can view the header information at the top of the file, which describes the dataset. We can see that there are 6 input parameters that quantify the processor cycle time and different memory parameters (we haven't covered channel width in class, but this is the width of the databus that can be used to talk to a cache or main memory in a computer). The last parameter is the relative performance of the CPU with these parameters. Looking at the rest of the file, we can see there are 209 training examples for different CPU configurations that provide is a complete set of parameters.

To load this data into Weka for analysis, select 'Explorer'. In the following screen select 'open file' and locate the `cpu.arff` file. After loading the file, we can see that there are 7 parameters and in the lower right hand corner, we can see statistical information about the input and output parameters.

To learn the weights for our linear regression, click the 'classify' tab. Press the 'classifier' button and select functions -> linear regression. The right of this box, we can see parameters related to how the weights are computed. Make sure decimal places is set to 4 or greater. To learn the weights, Weka partitions the data into training and test sets – for our training, we will use the default approach of creating 10 'folds' of data, where random permutations of the data are split into 10 pieces. 9 of these pieces are used for training, while 1 is used for test. These values are rotated, and the weights are learned that minimize error in the output, in this case 'relative CPU performance'.

You can find the weight values in the classifier output window as the values after 'class =' that multiply each parameter. These are the weights we will use for our matrix multiplication algorithm. Since we want to avoid the use of floating point values, multiply each of these values by 1000 and round them (if interested in learning more about fixed point math, more details can be found here: https://en.wikipedia.org/wiki/Fixed-point_arithmetic). After a result is computed using matrix multiplication with the weights, we can divide the result by 1000 to determine the final result.

Deliverables:

-Screenshot of Weka output that shows the weights computed when training using a 10-fold cross validation. Include the performance results, which are included after the linear function.

Implementing Matrix Multiplication in MARS

To understand the low level performance of the matrix multiplication algorithm for the MIPS architecture, your task is to implement this as a series of assembly instructions. The fixed point weights you computed in the previous section can be input directly into the MIPS starter code below. The 6 x 100 training examples from the .arff file are hard coded into the simulator's memory in the starter code in another memory location. These samples are identical to what you trained on in the .arff file – only the relative performance output is excluded.

It is your task to predict the relative performance using the weights provided from Weka and store the results in a fixed memory region. The place to put predictions is provided in the starter code. To compute these precictions, you need to perform matrix multiplication between the weight vector and the matrix of training examples.

For your reference, a C implementation of matrix multiplication is provided below and needs to be translated into MIPS assembly. We have provided the prologue and epilogue assembly code and have preinitialized the memory with training data.

C Code for Matrix Multiplication:

```
#include <stdio.h>

int main()
{
  int m, n, p, q, c, d, k, sum = 0;
  int first[10][10], second[10][10], multiply[10][10];

  printf("Enter number of rows and columns of first matrix\n");
  scanf("%d%d", &m, &n);
  printf("Enter elements of first matrix\n");

  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
      scanf("%d", &first[c][d]);

  printf("Enter number of rows and columns of second matrix\n");
  scanf("%d%d", &p, &q);

  if (n != p)
    printf("The matrices can't be multiplied with each other.\n");
  else
  {
    printf("Enter elements of second matrix\n");

    for (c = 0; c < p; c++)
```

```
        for (d = 0; d < q; d++)
          scanf("%d", &second[c][d]);

      for (c = 0; c < m; c++) {
        for (d = 0; d < q; d++) {
          for (k = 0; k < p; k++) {
            sum = sum + first[c][k]*second[k][d];
          }

          multiply[c][d] = sum;
          sum = 0;
        }
      }

      printf("Product of the matrices:\n");

      for (c = 0; c < m; c++) {
        for (d = 0; d < q; d++)
          printf("%d\t", multiply[c][d]);

        printf("\n");
      }
    }

    return 0;
  }
```

MIPS skeleton code:

```
   .data
N: .word  1  # number of input vectors x
M: .word  6 # number of features
P: .word  100  # weight vector is 1 dimensional
learned_weight_vector: .word  1, 2, 3, 4, 5, 6 #**your learned weights here**
x_data: .word
125,256,6000,256,16,128
29,8000,32000,32,8,32
29,8000,32000,32,8,32
29,8000,32000,32,8,32
29,8000,16000,32,8,16
26,8000,32000,64,8,32
23,16000,32000,64,16,32
23,16000,32000,64,16,32
23,16000,64000,64,16,32
23,32000,64000,128,32,64
400,1000,3000,0,1,2
400,512,3500,4,1,6
60,2000,8000,65,1,8
50,4000,16000,65,1,8
350,64,64,0,1,4
200,512,16000,0,4,32
167,524,2000,8,4,15
143,512,5000,0,7,32
143,1000,2000,0,5,16
110,5000,5000,142,8,64
```

```
143,1500,6300,0,5,32
143,3100,6200,0,5,20
143,2300,6200,0,6,64
110,3100,6200,0,6,64
320,128,6000,0,1,12
320,512,2000,4,1,3
320,256,6000,0,1,6
320,256,3000,4,1,3
320,512,5000,4,1,5
320,256,5000,4,1,6
25,1310,2620,131,12,24
25,1310,2620,131,12,24
50,2620,10480,30,12,24
50,2620,10480,30,12,24
56,5240,20970,30,12,24
64,5240,20970,30,12,24
50,500,2000,8,1,4
50,1000,4000,8,1,5
50,2000,8000,8,1,5
50,1000,4000,8,3,5
50,1000,8000,8,3,5
50,2000,16000,8,3,5
50,2000,16000,8,3,6
50,2000,16000,8,3,6
133,1000,12000,9,3,12
133,1000,8000,9,3,12
810,512,512,8,1,1
810,1000,5000,0,1,1
320,512,8000,4,1,5
200,512,8000,8,1,8
700,384,8000,0,1,1
700,256,2000,0,1,1
140,1000,16000,16,1,3
200,1000,8000,0,1,2
110,1000,4000,16,1,2
110,1000,12000,16,1,2
220,1000,8000,16,1,2
800,256,8000,0,1,4
800,256,8000,0,1,4
800,256,8000,0,1,4
800,256,8000,0,1,4
800,256,8000,0,1,4
125,512,1000,0,8,20
75,2000,8000,64,1,38
75,2000,16000,64,1,38
75,2000,16000,128,1,38
90,256,1000,0,3,10
105,256,2000,0,3,10
105,1000,4000,0,3,24
105,2000,4000,8,3,19
75,2000,8000,8,3,24
75,3000,8000,8,3,48
175,256,2000,0,3,24
300,768,3000,0,6,24
300,768,3000,6,6,24
300,768,12000,6,6,24
300,768,4500,0,1,24
```

```
300,384,12000,6,1,24
300,192,768,6,6,24
180,768,12000,6,1,31
330,1000,3000,0,2,4
300,1000,4000,8,3,64
300,1000,16000,8,2,112
330,1000,2000,0,1,2
330,1000,4000,0,3,6
140,2000,4000,0,3,6
140,2000,4000,0,4,8
140,2000,4000,8,1,20
140,2000,32000,32,1,20
140,2000,8000,32,1,54
140,2000,32000,32,1,54
140,2000,32000,32,1,54
140,2000,4000,8,1,20
57,4000,16000,1,6,12
57,4000,24000,64,12,16
26,16000,32000,64,16,24
26,16000,32000,64,8,24
26,8000,32000,0,8,24
26,8000,16000,0,8,16
480,96,512,0,1,1
203,1000,2000,0,1,5
predictions: .space 400
  .text
main:
    # Store M, N, P in $a? registers
    lw   $a0, N
    lw   $a1, M
    lw   $a2, P
    jal  multiply
    ori $v0,$0,10                        # end program gracefully
    syscall
multiply:
  # Register usage:
  # n is $s0, m is $s1, p is $s2,
  # r is $s3, c is $s4, i is $s5,
  # sum is $s6
  # Prologue
  sw   $fp, -4($sp)
  la   $fp, -4($sp)
  sw   $ra, -4($fp)
  sw   $s0, -8($fp)
  sw   $s1, -12($fp)
  sw   $s2, -16($fp)
  sw   $s3, -20($fp)
  sw   $s4, -24($fp)
  sw   $s5, -28($fp)
  sw   $s6, -32($fp)
  addi $sp, $sp, -36
  # Save arguments
  move $s0, $a0             # n
  move $s1, $a1             # m
  move $s2, $a2             # p
  li   $s3, 0               # r = 0
  li   $t0, 4               # sizeof(Int)
  ##############################
  # Your code here
  # ...
  ##############################

mult_end:
  # Epilogue
  lw   $ra, -4($fp)
  lw   $s0, -8($fp)
  lw   $s1, -12($fp)
  lw   $s2, -16($fp)
  lw   $s3, -20($fp)
  lw   $s4, -24($fp)
  lw   $s5, -28($fp)
  lw   $s6, -32($fp)
  la   $sp, 4($fp)
  lw   $fp, ($fp)
  jr   $ra
```

Deliverables:

-Using the results from Weka, compute Y for 3 training examples (i.e. multiply weights by 3 rows of data from Weka). Show all work.

-Provide a screenshot of the MARS memory map that shows the fixed point approximation for these 3 training examples. Comment on any error introduced by our fixed point approximation.

-Compare the predicted result to the ground truth relative performance value in the .arff file. How accurately did the linear regression function predict the value?

Optimizing Cache Performance Using a Cache Simulator

Matrix multiplication will be slow if we don't provide a cache organization that can ensure a) the weights used in matrix multiplication stay in the cache and b) the spatial locality of each set of input data is accounted for.

In MARS, it is simple to perform these cache simulations. Under 'tools' select 'Data Cache Simulator'. At the bottom left of this window, select 'connect to MIPS'. With the simulator connected, run your matrix multiplication code for a given cache configuration. In the cache performance section of this window, the tool will report the hit or miss rate.

For each cache simulation for a given organization, click the ‘reset’ button to reset the cache contents and the hit or miss statistics.

Deliverables:

-Try 12 different cache configurations that use different sizes, block lengths, and levels of associativity. Report the configuration results you tried in a table.
-What was the hit rate for these different configurations? Ensure that at least one of your configurations achieved $\geq$ 90% hit rate.
-Provide intuition into why each configuration achieves its performance.
-Provide a screenshot of the cache configuration that achieved $\geq$ 90% hit rate.